Written Examination, December 15th, 2021 Course no. 02157

The duration of the examination is 4 hours.

Course Name: Functional programming

Allowed aids: All written material

The problem set consists of 4 problems which are weighted approximately as follows:
Problem 1: 35%, Problem 2: 20%, Problem 3: 15%, Problem 4: 30%

Marking: 7 step scale.

In your programs you are allowed to introduce helper functions; but you must also provide a declaration for each of the required functions, so that it has exactly the type and effect asked for.

You are, in general, allowed to use the .NET library including the modules described in the textbook, e.g., List, Set, Map, Seq, etc. But be aware of the special condition stated in Problem 1.

You are not allowed to use imperative features, like assignments, arrays and so on, in your solutions.

A typo has been corrected in Problem 1 and one in Problem 4 as well.

 ... Continued on next page

Problem 1 (35%)

A club rewards members with *prizes* for *achievements* using a model captured by the following type declarations:

```
type Prize       = string
type Achievement = int
type PrizeTable = (Prize * Achievement) list
```

The kinds of club, prizes and achievements are not important; we just need to be able to compare achievements and we assume that they are described by integers.

A *prize table* is a list of pairs: $[(p_1, a_1); (p_2, a_2); \ldots; (p_n, a_n)]$, where p_i is a prize and a_i is an achievement. For a prize table we require that achievements occur in ascending order, that is, $a_1 < a_2 < \cdots < a_n$. This requirement is called the *prize-table invariant* or just *invariant*. The following example prize table satisfies the invariant:

```
let pt = [("p1", 3); ("p2", 5);("p3", 8); ("p4", 11) ]
```

The questions 1. to 5. in this problem should be solved without using functions from the libraries `List`, `Seq`, `Set` and `Map`. That is, the requested functions should be declared using explicit recursion.

1. Declare a function `inv: PrizeTable -> bool` that can check whether the invariant holds for a given prize table.

From now on you can assume that prize-table arguments to functions satisfy the invariant; but you must ensure that prize-tables returned by functions satisfy the invariant.

2. Declare a function `prizesFor: Achievement -> PrizeTable -> Prize list` so that `prizesFor` *a pt* is the list of prizes that are associated with achievements smaller that or equal to *a* in *pt*. For example, `prizesFor 7 pt` = `["p1"; "p2"]`.
3. Declare a function `increase: int -> PrizeTable -> PrizeTable`. The value of the expression `increase` *k pt* is the prize table obtained from *pt* by adding *k* to every achievement. For example, `increase 2 pt` = `[("p1",5);("p2",7);("p3",10);("p4",13)]`.
4. Declare a function `add: (Prize*Achievement) * PrizeTable -> PrizeTable`. The value of `add`$((p, a), pt)$ is the prize table obtained from *pt* by insertion of the pair (p, a). An exception should be raised if there is a pair in *pt*, for which the achievement is equal to a. For example, `[("p1",3);("p2",5);("p3",8);("p35",10);("p4",11)]` is the value of `add(("p35",10),pt)`.
5. Declare a function `merge` pt_1 pt_2 that gives the prize table consisting of all the pairs from prize table pt_1 and prize table pt_2. If a pair (p_1, a_1) in pt_1 has the same achievement as a pair (p_2, a_2) in pt_2, i.e. $a_1 = a_2$, then an exception should be raised.

 ... Continued on next page

In the last question you may use the following functions from the `List` library: `filter`, `map`, `fold` and `foldBack`.

6. Give alternative declarations for the functions `prizesFor`, `increase` and `merge` from questions 2., 3. and 5. You may use the above-mentioned functions from the `List` library and other functions from this problem; but you should not use explicit recursion in the declarations.

Problem 2 (20%)

The function `choose` from the `List` library could have the following declaration:

```
let rec choose f xs =
   match xs with
   | []      -> []                              (* C1 *)
   | x::rest -> match f x with
                | None   -> choose f rest       (* C2 *)
                | Some y -> y::choose f rest;;  (* C3 *)
val choose : ('a -> 'b option) -> 'a list -> 'b list
```

Notice that the F# system automatically infers the type of `choose`.

1. Give an argument showing that `('a -> 'b option) -> 'a list -> 'b list` is the most general type of `choose`. That is, any other type for `choose` is an instance of `('a -> 'b option) -> 'a list -> 'b list`.

let `chEven` be declared by:

```
let chEven n = if n%2=0 then Some (string n) else None;;
```

2. Give an evaluation of the expression `choose chEven [1;2;3;4;5]`. Use the notation $e_1 \rightsquigarrow e_2$ from the textbook and include at least as many steps as there are recursive calls.

The declaration of `choose` is not tail recursive.

3. Provide a declaration of a tail-recursive variant of `choose` that is based on an accumulating parameter. Your tail-recursive declaration must be based on an explicit recursion.
4. Provide a declaration of a tail-recursive variant of `choose` that is continuation-based. Your tail-recursive declaration must be based on an explicit recursion.

 ... Continued on next page

Problem 3 (15%)

Consider the following declarations:

```
type T = | One of int | Two of int * T * int * T

let rec f p t =
   match t with
   | One v when p v           -> [v]             (* C1 *)
   | Two(v1,t1,_,_) when p v1 -> v1::f p t1      (* C2 *)
   | Two(_,_,v2,t2)           -> v2::f p t2      (* C3 *)
   | _                        -> [];;            (* C4 *)
```

1. Give the type for `f` and describe what `f` computes. Your description should focus on what it computes, rather than on individual computation steps.

Notice that the declaration of `f` has a match expression with 4 clauses marked `C1` to `C4` in comments.

A *test description* for `f` consists of

- a value p_v for argument `p`,
- a value t_v for argument `t`,
- the expected value of `f` p_v t_v, and
- an enumeration of the clauses that are selected during evaluation of `f` p_v t_v. The order in which clauses are enumerated is not significant. Repeated enumeration of a clause is not necessary.

2. Give a small number (≤ 4) of test descriptions for `f`. Together they should ensure that every clause of `f` is selected during an evaluation.

 ... Continued on next page

Problem 4 (30%)

A type for so-called *tries* is defined as a tree type `Trie<'a>`, where a node carries a value of type `'a`, a truth value, and an arbitrary number of child tries:

```
type Trie<'a>    = N of 'a * bool * Children<'a>
and Children<'a> = Trie<'a> list
```

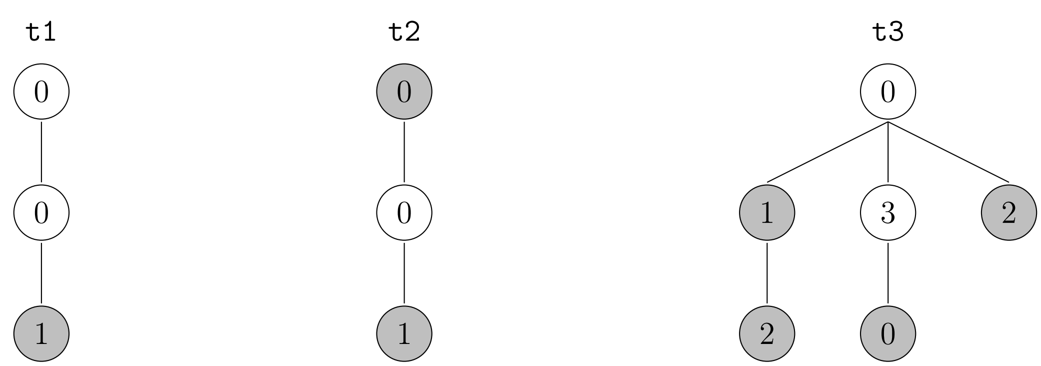
Consider the three values `t1`, `t2` and `t3` of type `Trie<int>`:

```
let t1 = N(0, false, [N(0, false, [N(1,true,[])])]);;

let t2 = N(0, true, [N(0, false, [N(1,true,[])])]);;

let ta = N(1,true,[N(2,true,[])]);;
let tb = N(3,false,[N(0,true,[])]);;
let tc = N(2,true,[]);;
let t3 = N(0,false, [ta;tb;tc]);;
```

The three values are illustrated as trees in the following figure, where each node carry an integer value, and a shaded node indicates that the truth value associated with the node is `true`. Shaded nodes are also called *accepting nodes*.



`t1` accepts [0;0;1]   `t2` accepts [0] and [0;0;1]   `t3` accepts [0;1], [0;1;2], [0;3;0] and [0;2]

A value in a node of a trie is called a *letter*. For example, trie `t3` contains four letters: $0, 1, 2, 3$.

A *word* is a list of letters. Furthermore, a word w is *accepted by* a trie t if there is a path from the root of t to an accepting node, so that w equals the list of letters of the nodes of the path. For example, $[0; 1; 2]$ is accepted by `t3` and the tries `t1`, `t2` and `t3` accept 1, 2 and 4 words, respectively, as shown in the figure.

 ... Continued on next page

1. Declare a function that counts the number of nodes of a trie. For example, `t3` has 6 nodes.
2. Declare a function `accept` w t that can check whether word w is accepted by trie t. Give the type of `accept`.
3. Declare a function `wordsOf: Trie<'a> -> Set<'a list>` that gives the set of words accepted by a trie t.

Leaves of tries have the form $\texttt{N}(v, b, [])$. Leaves where $b = $ `false` do not contribute to the words accepted by a trie and such leaves are called *useless*.

4. Declare a function that can check whether a trie contains useless leaves.

The *degree of a node* $\texttt{N}(v, b, ts)$ is the length of the list of children ts. The maximum degree of all nodes in a trie is called the *degree of a trie*.

5. Declare a function that computes the degree of a trie.